IMPLEMENTACION DE UNA COMPUTADORA CAPAZ DE EJECUTAR INSTRUCCIONES RV32I

ALARCON FAJARDO, ROCIO BELEN

INGENIERIA ELECTRONICA

2025

RESUMEN:

En este trabajo se desarrolló e implementó una computadora capaz de ejecutar instrucciones del conjunto RV32I de la arquitectura RISC-V. El proyecto incluyó el diseño del datapath, la unidad de control como máquina de estados finitos y la descripción en VHDL de todos los módulos necesarios para su funcionamiento. Posteriormente, el sistema fue simulado para verificar su correcto comportamiento y finalmente sintetizado e implementado en la placa EDU-CIAA-FPGA. Los resultados obtenidos en simulación y en hardware real confirmaron el correcto funcionamiento del procesador diseñado.

INTRODUCCION:

• Definir Arquitectura de Computadora:

La arquitectura de computadora es la especificación de los atributos visibles para el programador que determinan cómo se ejecutan los programas en una máquina, incluye principalmente, el conjunto de instrucciones (ISA), los registros, los tipos y tamaños de datos, los modos de direccionamiento, el modelo de memoria y el comportamiento observable de operaciones como saltos, interrupciones y excepciones. En otras palabras, describe qué hace el computador desde el punto de vista funcional y qué "contrato" ofrece al software, independientemente de cómo esté implementado internamente (Harris & Harris, 2022).

• Explicar brevemente de que se trata la arquitectura RISC-V y el conjunto de instrucciones RV32I

La arquitectura RISC-V es una arquitectura de computadora basada en la filosofía RISC (Reduced Instruction Set Computer) es decir, un diseño con un conjunto de instrucciones simple y regular, pensado para que la implementación del hardware sea eficiente y escalable. RISC-V se caracteriza por ser una ISA abierta y modular, define un conjunto base obligatorio y permite agregar extensiones estándar (por ejemplo, para multiplicación, punto flotante, etc.) según las necesidades del sistema. En esa idea, el software ve un "contrato" claro (la ISA) y el hardware puede implementarse de muchas maneras distintas manteniendo compatibilidad (Harris & Harris, 2022).

El conjunto de instrucciones RV32I es el ISA base de RISC-V para procesadores de 32 bits ("RV32") con el conjunto entero ("I", de Integer). Define las operaciones mínimas necesarias para ejecutar programas, operaciones aritméticas y lógicas sobre registros

(por ejemplo suma, AND, OR), cargas y almacenes para mover datos entre memoria y registros (load/store), saltos y bifurcaciones para control de flujo (jumps/branches), y soporte para instrucciones de control básico del sistema. Es "base" porque sobre RV32I se pueden construir procesadores completos y, si se requiere más funcionalidad o rendimiento, se le suman extensiones manteniendo el núcleo compatible (Harris & Harris, 2022).

• Describir el conjunto de registros de la arquitectura RISC-V

Las instrucciones RV32I tienen un formato fijo de 32 bits y se organizan en seis tipos, R, I, S, B, U y J (*véase Tabla 1*). Cada tipo distribuye los campos opcode, rd, rs1, rs2, funct3, funct7 e inmediato de manera distinta según la función que cumpla la instrucción. El tipo R se utiliza para operaciones aritmético - lógicas entre registros y codifica la operación mediante la combinación de funct3 y funct7.

- El tipo I se emplea para operaciones con inmediato y cargas desde memoria; en este caso el inmediato ocupa los bits más significativos y se extiende con signo.
- El tipo S corresponde a las instrucciones de almacenamiento y construye el inmediato a partir de dos campos separados.
- El tipo B se usa en saltos condicionales y arma el inmediato concatenando bits dispersos, agregando un cero menos significativo porque el desplazamiento está alineado.
- El tipo U carga inmediatos de 20 bits en la parte alta del registro.
- El tipo J se emplea para saltos incondicionales con enlace y también reconstruye el inmediato a partir de campos no contiguos.

En cuanto al cálculo de los inmediatos, en los tipos I y S se realiza extensión de signo directa del valor reconstruido. En los tipos B y J el inmediato representa un desplazamiento relativo al PC y se obtiene concatenando los bits indicados en el formato y agregando un bit cero al final. En el tipo U el inmediato se coloca en los 20 bits superiores del registro desplazándolo 12 posiciones hacia la izquierda.

Para el opcode decimal 19 (0010011) correspondiente a instrucciones ALU con inmediato, la operación se determina principalmente por el campo funct3 y, en los desplazamientos, también por funct7. Para el opcode decimal 51 (0110011) que corresponde a operaciones ALU entre registros, la operación matemática queda completamente definida por la combinación funct3 + funct7 (por ejemplo, ADD y SUB se distinguen por funct7).

Finalmente, el opcode decimal 99 (1100011) corresponde a saltos condicionales. En este caso la ALU se utiliza para comparar rs1 y rs2. Para BEQ y BNE se realiza una resta y se evalúa la señal de cero; el salto se toma si la condición indicada por funct3 se cumple. Para BLT, BGE y sus versiones unsigned, la ALU evalúa la comparación correspondiente. Si la condición es verdadera, el PC se actualiza con el desplazamiento inmediato en caso contrario, continúa con PC + 4. (Harris & Harris, 2022).

Tabla 1: Conjunto de instrucciones RV32I (Harris & Harris, 2022)

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

• Definir Microarquitectura de computadora

Si la arquitectura de una computadora es el diseño lógico y abstracto (el "qué"), la microarquitectura es la implementación física y concreta de ese diseño (el "cómo"). Se define como la organización específica de los bloques constructivos digitales (como registros, ALUs, multiplexores, memorias y máquinas de estado) conectados de tal manera que puedan ejecutar las instrucciones definidas por la arquitectura.

En otras palabras, la microarquitectura es el diseño del hardware interno del procesador. Mientras que la arquitectura (ISA) es un contrato fijo que asegura la compatibilidad del software, la microarquitectura es flexible y varía entre diferentes procesadores. Esto permite que existan múltiples implementaciones de una misma arquitectura (como la familia de procesadores x86 de Intel o AMD) que corren el mismo software, pero que difieren radicalmente en rendimiento, consumo de energía, área de silicio y costo. (Harris & Harris, 2022).

Específicamente, una microarquitectura se compone de dos partes fundamentales que interactúan entre sí:

El Datapath (Ruta de Datos): Contiene los elementos funcionales que operan sobre los datos (bancos de registros, unidades aritméticas, buses de interconexión).

La Unidad de Control: La lógica que dirige el flujo de datos a través del datapath, indicando qué operaciones realizar en cada momento según la instrucción que se esté ejecutando.

DESARROLLO:

Datapath Implementado:

El datapath diseñado implementa una microarquitectura compatible con el conjunto de instrucciones RV32I (*véase figura 1*). Está compuesto por los bloques principales de un procesador: contador de programa (PC), registro de instrucción (IR), banco de registros, ALU de 32 bits, memoria y la lógica de selección mediante multiplexores.

El PC almacena la dirección de la instrucción actual y puede actualizarse de manera secuencial (PC + 4) o con una dirección de salto calculada. Esta selección se realiza mediante un multiplexor controlado por la unidad de control. El incremento por 4 permite la ejecución normal del programa cuando no hay saltos.

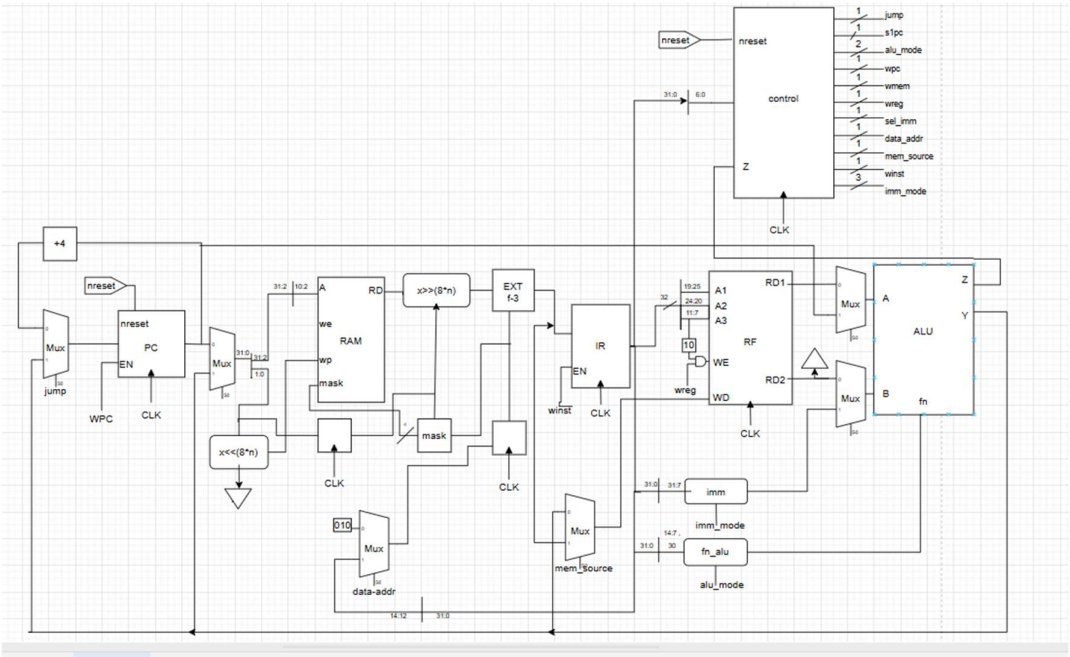
La memoria se utiliza tanto para instrucciones como para datos. Durante la etapa de búsqueda, el PC direcciona la memoria para obtener la instrucción, que luego se almacena en el registro de instrucción (IR). Desde el IR se extraen los campos necesarios (opcode, funct3, funct7, rs1, rs2, rd) para alimentar el banco de registros y la unidad de control.

El banco de registros permite leer dos operandos simultáneamente y escribir un resultado. La escritura se habilita con la señal wreg, y el dato puede provenir de la ALU, de memoria o del valor PC+4 en el caso de saltos con enlace.

La ALU realiza las operaciones aritméticas y lógicas definidas por el conjunto RV32I. El segundo operando puede ser un registro o un inmediato extendido, dependiendo del tipo de instrucción. Además, la ALU genera la señal de condición utilizada para decidir saltos condicionales.

El generador de inmediatos reconstruye el valor inmediato según el formato de la instrucción (I, S, B, U o J), aplicando extensión de signo cuando corresponde.

Finalmente, la unidad de control, implementada como una máquina de estados, genera las señales necesarias para coordinar la búsqueda, decodificación y ejecución de las instrucciones.



Módulos implementados:

El procesador fue desarrollado siguiendo una arquitectura modular, separando cada bloque funcional en una entidad independiente para facilitar la organización, simulación y síntesis del sistema.

- El módulo **cpu.vhd** implementa el datapath principal del procesador. Integra el contador de programa, el registro de instrucción, el banco de registros, la ALU y las señales de interconexión necesarias para ejecutar las instrucciones.
- El módulo **control.vhd** implementa la máquina de estados finitos (FSM) encargada de generar las señales de control del procesador. Analiza el opcode y las señales de condición provenientes de la ALU para determinar la secuencia de ejecución de cada instrucción.
- El módulo **alu.vhd** realiza las operaciones aritméticas y lógicas de 32 bits. La operación se selecciona mediante una señal de control proveniente de la unidad de control y genera, además del resultado, señales de condición utilizadas en las instrucciones de salto.
- El módulo **registro_32x32.**vhd implementa el banco de registros del procesador, con dos puertos de lectura y un puerto de escritura, permitiendo almacenar los operandos y resultados de las operaciones.
- El módulo **ram_512x32.vhd** implementa la memoria SRAM de 512 palabras de 32 bits, utilizada tanto para instrucciones como para datos.
- El módulo **ram_controller.vhd** administra el acceso a la memoria, controlando las operaciones de lectura y escritura según el tipo de instrucción ejecutada.
- El módulo **crossbar.vhd** gestiona la interconexión entre el procesador, la memoria y los perifericos, permitiendo direccionar correctamente los accesos de datos.

- El módulo **O_controller.vhd** implementa la interfaz de salida del sistema, permitiendo manejar los 8 bits de salida definidos en la consigna.
- El módulo **reset_al_inicializar_fpga.vhd** genera la señal de reset al iniciar la FPGA, asegurando que el sistema comience en un estado conocido.
- El módulo **top.vhd** integra todos los bloques anteriores, conformando el microcontrolador completo y permitiendo su síntesis y carga en la placa EDU-CIAA-FPGA.

RESULTADOS:

La figura 2 presenta la máquina de estados finitos (FSM) encargada de generar las señales de control del procesador (*véase Figura 2*). Esta unidad coordina la secuencia de búsqueda, decodificación y ejecución de las instrucciones RV32I, activando en cada estado las señales correspondientes del datapath.

El funcionamiento comienza en el estado INICIO donde se establecen los valores por defecto de las señales. A continuación, el sistema pasa al estado LEE_MEM_PC en el cual se accede a memoria utilizando el valor del PC para obtener la instrucción. Posteriormente, en el estado CARGA_IR la instrucción leída se almacena en el registro de instrucción.

En el estado DECODIFICA, la unidad de control analiza el campo opcode y determina el camino a seguir. Según el tipo de instrucción, la FSM transita hacia estados específicos de ejecución:

- Para instrucciones tipo R (opcode 0110011), se ejecuta el estado EJECUTA_R, habilitando la escritura en el banco de registros.
- Para instrucciones tipo I (0010011), se ejecuta EJECUTA_I, configurando la ALU para operar con inmediato.
- Para instrucciones de carga (LOAD), primero se calcula la dirección efectiva y luego se realiza la lectura desde memoria, finalizando con la escritura en el registro destino.
- Para instrucciones de almacenamiento (STORE), se calcula la dirección y se habilita la escritura en memoria.
- Para saltos condicionales (BRANCH), el estado EVAL_BRANCH evalúa la señal de condición generada por la ALU y decide si actualizar el PC con el desplazamiento correspondiente.
- Para instrucciones JAL y JALR, se activa la señal de salto y se escribe la dirección de retorno en el registro destino.
- Para LUI y AUIPC, se realiza la carga del inmediato según el formato correspondiente.

Una vez completada la ejecución de cada instrucción, la FSM retorna al estado de búsqueda, repitiendo el ciclo.

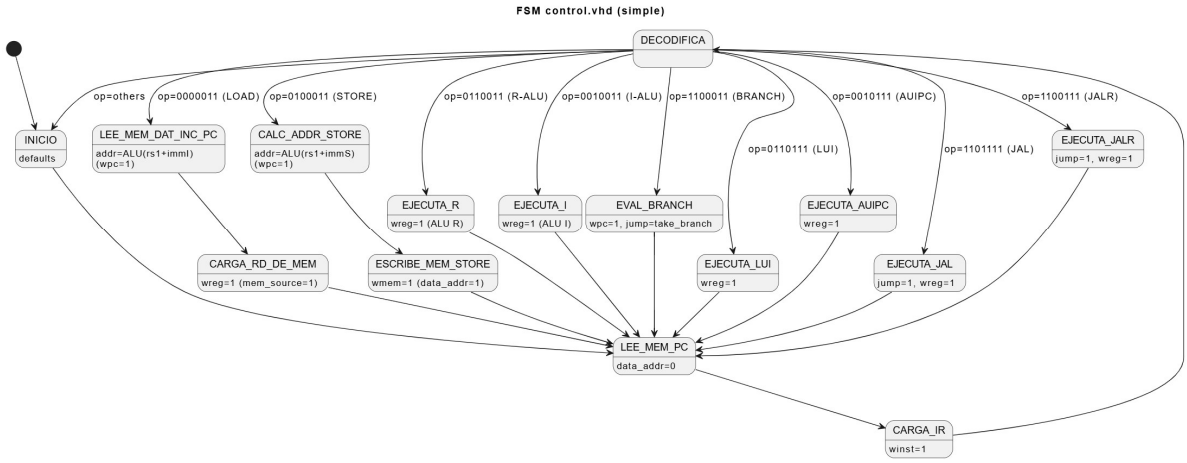


figura 1: diagrama de estados de control de la CPU.

Simulación:

Para verificar el correcto funcionamiento y la estabilidad del procesador RISC-V, se diseñaron y ejecutaron cuatro rutinas de prueba en lenguaje ensamblador. A continuación, se detallan estos casos de estudio. Para cada prueba se presenta una Tabla que especifica el código de máquina, las instrucciones ejecutadas y su propósito técnico; seguida por una Figura con el diagrama de tiempos obtenido mediante simulación. En estas capturas se evidencia el comportamiento sincronizado del hardware, observando la evolución de señales clave como el contador de programa (pc), el registro de instrucción (ir) y la salida de la ALU (alu_y), lo que confirma la correcta ejecución de cada algoritmo ciclo a ciclo.

1° prueba: Suma de dos números inmediatos y almacenamiento en memoria.

Tabla 2: Código de máquina e instrucciones RISC-V para la suma de dos números

Código Máquina (Hex)	**Instrucción RISC-V**	**Formato**	**Explicación Técnica**
00500093	ADDI x1, x0, 5	Tipo I	Carga el valor inmediato 5 en el registro x1.
00400113	ADDI x2, x0, 4	Tipo I	Carga el valor inmediato 4 en el registro x2.
002081b3	ADD x3, x1, x2	Tipo R	Suma x1 (5) + x2 (4) y guarda el resultado (9) en x3.
00302023	SW x3, 0(x0)	Tipo S	Guarda el valor de x3 (9) en la dirección de memoria 0.

0000006f	JAL x0, 0	Tipo J	Salto incondicional a la misma dirección (PC + 0). Es un bucle infinito para detener el programa.

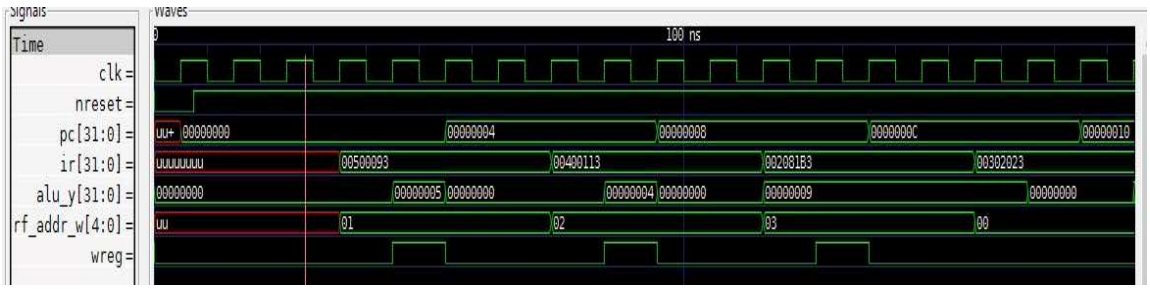


figura 2: diagrama de tiempos y señales de control durante la ejecucion de la prueba 1:

2° prueba: Evaluación de salto condicional y bifurcación de flujo.

Tabla 3: codigo de maquina e instrucciones RISC-V para la prueba de salto condicional.

Código Máquina (Hex)	Instrucción RISC-V	Formato	Explicación Técnica
00500093	ADDI x1, x0, 5	Tipo I	Inicializa x1 con el valor 5.
00500113	ADDI x2, x0, 5	Tipo I	Inicializa x2 con el valor 5.
00208263	BEQ x1, x2, 4	Tipo B	Compara si x1 == x2. Como 5 es igual a 5, salta.
deadc1b7	LUI x3, 0xdeadc	Tipo U	Carga la constante mágica superior. x3 se convierte en 0xDEADC000.
00100193	ADDI x3, x0, 1	Tipo I	Sobrescribe x3 con el valor 1.
0000006f	JAL x0, 0	Tipo J	Bucle infinito. El PC se queda apuntando a esta misma dirección para siempre.

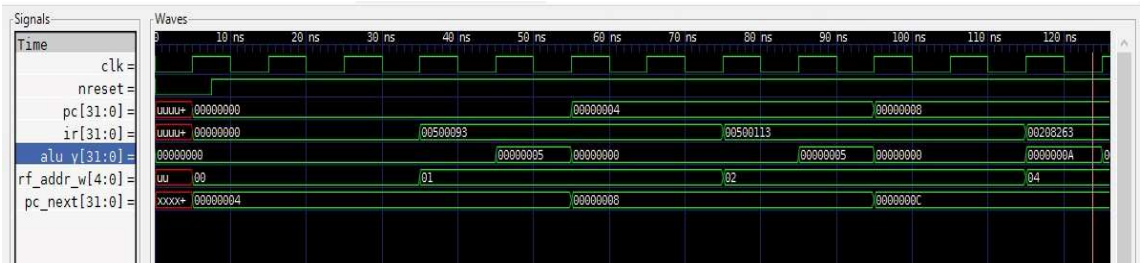


figura 3: Diagrama de tiempos y alternacion de contador de programa durante la prueba 2.

3° prueba: Operaciones aritméticas (resta) y lógicas a nivel de bits (OR, AND)

Tabla 4: codigo de maquina e instrucciones RISC-V para operaciones de resta y logicas.

Código Máquina (Hex)	**Instrucción RISC-V**	**Formato**	**Explicación Técnica**
00f00093	ADDI x1, x0, 15	Tipo I	Carga el valor 15 (0xF) en el registro x1.
00500113	ADDI x2, x0, 5	Tipo I	Carga el valor 5 en el registro x2.
402081b3	SUB x3, x1, x2	Tipo R	Resta: x1 (15) - x2 (5). El resultado 10 se guarda en x3.
0020e233	OR x4, x1, x2	Tipo R	Operación lógica OR: 15 OR 5. 1111 OR 0101 = 1111 (15). Se guarda en x4.
0020f2b3	AND x5, x1, x2	Tipo R	Operación lógica AND: 15 AND 5. 1111 AND 0101 = 0101 (5). Se guarda en x5.
0000006f	JAL x0, 0	Tipo J	Bucle infinito (Trap). Detiene el programa aquí.

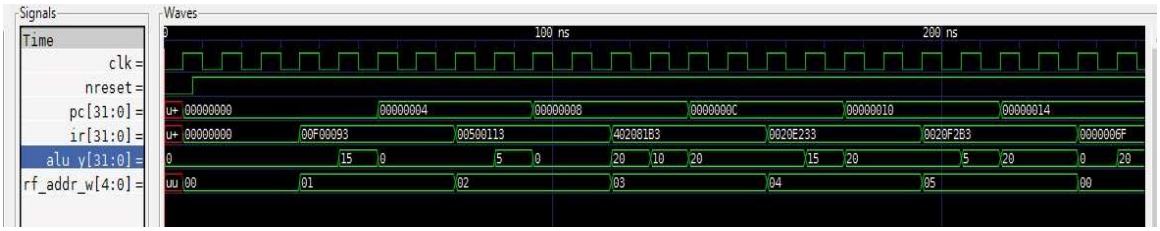


figura 4: diagrama de tiempos y respuestas de la ALU durante la prueba 3.

4° prueba

Tabla 5: codigo de maquina e instrucciones RISC-V para pruebas de acceso a memoria.

Código Máquina (Hex)	**Instrucción RISC-V**	**Formato**	**Explicación Técnica**
03700093	ADDI x1, x0, 55	Tipo I	Carga el valor inmediato 55 (0x37) en el registro x1.
04102023	SW x1, 64(x0)	Tipo S	Escritura en RAM: Guarda el valor de x1 (55) en la dirección de memoria 64 (0x40).

04002103	LW x2, 64(x0)	Tipo I	Lectura de RAM: Lee el dato que está en la dirección de memoria 64 y lo guarda en el registro x2.
0000006f	JAL x0, 0	Tipo J	Bucle infinito. Detiene el programa aquí.

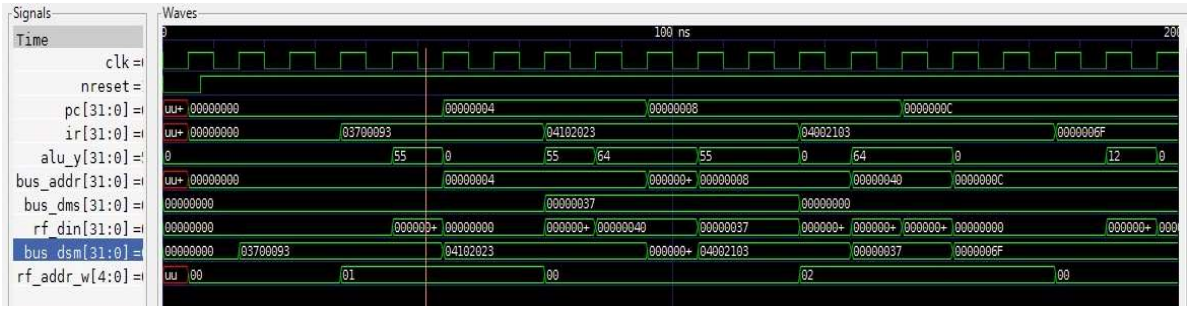


figura 5: diagrama de tiempos de las operaciones de lectura y escritura en memoria durante la prueba 4.

CONCLUSIONES:

A lo largo de este trabajo se logró diseñar e implementar una computadora funcional capaz de ejecutar instrucciones RV32I, integrando de forma concreta los conceptos de arquitectura y microarquitectura estudiados en clase. El diseño modular permitió organizar el sistema en bloques bien definidos, facilitando tanto la simulación como la posterior síntesis.

Las simulaciones realizadas mostraron que el procesador ejecuta correctamente operaciones aritméticas, lógicas, accesos a memoria y saltos condicionales. El análisis de las señales internas confirmó que la secuencia de estados y las señales de control actuaban como se esperaba en cada instrucción.

Tambien se realizó la implementación del módulo top en la placa EDU-CIAA-FPGA, utilizando el esquema de integración proporcionado por el profesor, permitió comprobar que el sistema también funciona correctamente en hardware real. El comportamiento observado coincidió con lo obtenido en simulación, lo que valida el diseño desarrollado.

En conclusión, el proyecto permitió comprender de manera práctica cómo una arquitectura como RISC-V puede materializarse en un sistema digital real, fortaleciendo tanto los conocimientos teóricos como las habilidades de diseño en VHDL.

BIBLIOGRAFIA:

Sarah L. Harris y Daivid Harris (2022). Digital Design and Computer Architecture, RISC-V Edition. Morgan Kaufmann.